A Practical and Efficient Method for Compare-Point Matching

Demos Anastasakis, Robert Damiano, Hi-Keung Tony Ma, Ted Stanion
Synopsys Inc.
{demosa, robertd, tonyma, teds}@synopsys.com

ABSTRACT

An important step in using combinational equivalence checkers to verify sequential designs is identifying and matching corresponding compare-points in the two sequential designs to be verified. Both non-function and function-based matching methods are usually employed in commercial verification tools. In this paper, we describe a heuristic algorithm using ATPG for matching compare-points based on the functionality of the combinational blocks in the sequential designs. Results on industrial-sized circuits show our methods are both practical and efficient.

Categories and Subject Descriptors

J.6 [**Computer-aided engineering**]: Verification, compare-point matching.

General Terms

Algorithms, Experimentation, Verification.

Keywords

Combinational verification, equivalence checking, latch mapping.

1. INTRODUCTION

Using combinational equivalence checkers to prove or disprove the functional equivalence of two sequential designs is a common and practical verification approach in today's design methodology. This verification approach is especially applicable when only combinational synthesis techniques are used for optimization. Corresponding combinational blocks from two sequential designs (one called the reference and the other the implementation) can be compared and verified using combinational formal techniques [1][2][3]. The sequential design can be a gate-level netlist or a register transfer level (RTL) source. No simulation vectors are required and the verification is complete over the entire vector space of the combinational blocks being verified. As designs become larger and more complex, verification methods utilizing combinational equivalence checkers and hence offering fast turnaround verification times and complete verifications, are rapidly gaining ground on the traditional simulation-based verification approaches.

An important step in applying combinational equivalence checkers is identifying and matching corresponding compare-points in the two sequential designs to be verified. A compare-point in a sequential design is a combinational logic endpoint during verification. A compare-point can be an output port, register, latch or black-box input pin. Compare-point matching (CPM) methods in commercial verification tools can be broadly divided into two classes: non-function-based and function-based.

Non-function-based methods use name or structural comparisons to match compare-points in the sequential designs. In most production verification flows, a large percentage of compare-points are usually matched using non-function-based methods. A significant number of compare points usually are left unmatched due to design transformations that do not preserve signal names or dramatically modify the circuit structure of some portion of the designs. Function-based methods are the only viable automatic methods for matching the remaining compare-points without requiring non-trivial effort from the designers.

Various function-based techniques [4][5][6][7] have been proposed to identify latch correspondences between two sequential designs. They are all exact methods based on fixed-point computations. Given *N* latches in each sequential design, there are *N!* possible combinations to match them. All these exact methods in the worst case may have to enumerate all combinations, although in an implicit way. This is because they actually try to prove functional equivalence between latches in the two sequential designs.

Our function-based heuristic method does not attempt to prove functional equivalence between compare-points. It rather tries to establish inequivalence relationships between compare-points. The inequivalence information is then used to group compare-points that are most likely to be equivalent into pairs. Not only is this method practical and efficient for industrial-sized circuits, it also separates cleanly the compare-point matching process from the equivalence verification process that follows. A similar approach is described in [8], but it only uses random pattern simulation to discover inequivalence relationships rather than using an automatic test pattern generation (ATPG) program. It also does not handle cases where don't care conditions must be considered to show equivalence.

This paper is organized as follows. In Section 2, we first define the compare-point-matching (CPM) problem and describe how it can be solved as a partitioning problem. We then describe how we can use an ATPG program in conjunction with a search model to find solutions for the partitioning problem. Taken into consideration of don't cares, that are typical in the specification description (e.g. RTL source), to show equivalence between two designs are described in Section 3. Experimental results are presented in Section 4 and the paper is concluded in Section 5.

2. COMPARE-POINT MATCHING

2.1 Compare-Point-Matching Problem

We are given two sequential circuits; one called the reference (C_{ref}) and the other the implementation (C_{imp}). Without loss of generality, we assume the two circuits have the same number of latches N and the only compare-points considered are the latches. The CPM problem is to identify N latch-pairs such that each pair consists of one latch from C_{ref} and one from C_{imp} and the two latches are combinationally equivalent. The next-state functions of the latches in a latch-pair represent two compare-points to be verified. The outputs of the latches in a latch-pair are used as pseudo primary inputs that are always set to identical values during combinational verification of downstream compare-points. There are $N!$ possible way of creating the N latch pairs.

Most exact function-based methods [4][5][6][7] solve the compare-point matching and equivalence checking processes together and implicitly enumerate all $N!$ combinations. Our heuristic method on the other hand only tries to group latches that are most likely to be equivalent and let the equivalence checker that follows perform the verification task. Our heuristic method can also be used as an efficient front-end to any exact method.

2.2 Compare-Point Partitioning

We can identify and use inequivalence relationships between reference and implementation latches to solve the CPM problem. If we can show a reference latch in C_{ref} is not equivalent to $(N - 1)$ implementation latches in C_{imp}, then we can conclude it is most likely to be equivalent to the remaining implementation latch and hence group the two into a compare-point pair.

Theorem 1: Let C_{ref} and C_{imp} be two sequential circuits each containing N latches, and there exists a unique pairing of latches, one from each circuit, such that latches in each pair are combinationally equivalent. If it can be shown for each latch in C_{ref} there are $(N - 1)$ latches in C_{imp} that are not equivalent to it, then there is only one way left to group the latches into combinationally equivalent pairs.

Based on Theorem 1, we can devise an algorithm, similar to [6], to detect inequivalence between two latches, one at a time and eventually arrive at the final solution representing the unique grouping if it exists. The number of inequivalence checks can be exceedingly large, even though using functional or sequential random pattern simulation can help to reduce it. The inequivalence checking process can also inadvertently invoke a number of equivalence proofs.

The main idea behind our heuristic method is try to generate as much inequivalence information about latch pairs as possible in a single checking step. This is done by grouping potentially equivalent latches (compare-points) into a single partition and successively dividing each partition into two sub-partitions until all partitions contain only two latches, one from C_{ref} and one from C_{imp}. The heuristic method also makes use of the newly generated inequivalence information to facilitate the division process.

Let us assume we are given two sequential circuits, C_{ref} and C_{imp}, and M partitions of potentially equivalent latches. Each partition contains equal number of latches from C_{ref} and C_{imp}. A reference latch in a partition is potentially equivalent to any implementation latch in the same partition and is already proved to be inequivalent to any implementation latch in any other partition. If M is equal to 1, then any latch in C_{ref} is potentially equivalent to any latch in C_{imp}. M may be greater than 1 at the beginning if non-function-based CPM process has been run and hence some initial equivalent and inequivalent relationship information is already obtained.

Let us choose a partition P_i and assume it contains $2N$ latches. Let the next-state functions of the reference latches in the partition be denoted by RNS_1 to RNS_n and the next-state functions of the implementation latches be denoted by INS_1 to INS_n. The next-state functions of the latches correspond to compare-points to be matched. Let the outputs of the reference latches in the partition be denoted by RPS_1 to RPS_n and the outputs of the implementation latches be denoted by IPS_1 to IPS_n. The outputs of the latches correspond to their present-states and represent pseudo primary inputs used to verify the downstream compare-points combinationally.

Let us create a search model circuit with outputs RNS_1 to RNS_n and INS_1 to INS_n corresponding to the next-state functions of the reference and implementation latches in the partition respectively. Without loss of generality, let us assume the next-state functions only depend on the primary inputs I_1 to I_p and pseudo primary inputs RPS_1 to RPS_n and IPS_1 to IPS_n. Figure 1 shows the search model circuit.

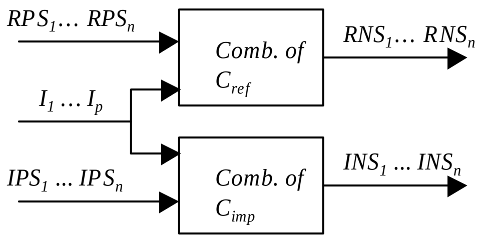


Fig. 1 Search Model Circuit.

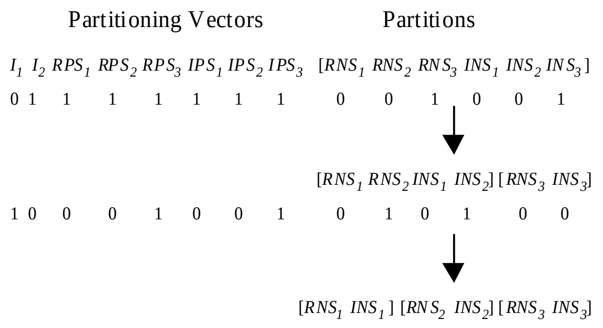
Theorem 2: Assume there exists some pairings of latches in P_i such that latches in each pair are combinationally equivalent. Given two latches L_{ref} and L_{imp} in P_i, if we can find a test vector ($I_1 \ldots I_p, RPS_1 \ldots RPS_n, IPS_1 \ldots IPS_n$) with the constraints $RPS_1 = \ldots = RPS_i = \ldots = RPS_n = IPS_1 = \ldots = IPS_i = \ldots = IPS_n$ that shows the two latches are inequivalent, then L_{ref} and L_{imp} can never be shown equivalent with any latch-pairing combinations.

Based on Theorem 2, we can devise a heuristic CPM method using successive partitioning to create potentially equivalent latch-pairs. Figure 2 shows the partitioning procedure. Whenever a test vector is found that sets opposite values between some RNS_i's and some INS_i's, we can divide a partition into two, one containing RNS_i's and INS_i's with values 0 and the other containing RNS_i's and INS_i's with values 1. Reference latches in one sub-partition cannot be equivalent to any implementation latches in the other sub-partition. In general, the search model is driven by pseudo primary inputs coming from multiple partitions. The test vectors are constrained to have RPS_i's and IPS_i's from the same partition having identical logic values.

```
While (unprocessed partition with size > 2 still exits) {
   pick a partition;
   search for a test vector that sets opposite values at
     some reference and implementation latches;
   if  (found such test vector) {
      simulate the test vector;
      create two new partitions, one containing latches
      with 0 values and the other with 1 values;
   } else {
      mark partition as processed and cannot be divided;
   }
}
```

Fig. 2 The Partitioning procedure.

An example to illustrate the working of the partitioning procedure is shown in Figure 3. The example has 2 inputs, I_1 and I_2, and six compare-points RNS_1, RNS_2, RNS_3, INS_1, INS_2 and INS_3. The pseudo primary inputs corresponding to the present states of the latches are RPS_1, RPS_2, RPS_3, IPS_1, IPS_2 and IPS_3. Initially, all compare-points are in a single partition. A partitioning vector consists of inputs I_1, I_2, RPS_1, RPS_2, RPS_3, IPS_1, IPS_2 and IPS_3. First, a vector (0,1,1,1,1,1,1,1), constrained with identical logic values at RPS_1, RPS_2, RPS_3, IPS_1, IPS_2 and IPS_3, is generated using random pattern generation or an ATPG program. This vector divides the single partition into 2. Partition [RNS_3 INS_3] consists of only two compare-points and is designated as a compare-point pair. The other partition is further divided into two sub-partitions with 2 elements by another vector (1,0,0,0,1,0,0,1) that is constrained with identical logic values at RPS_1, RPS_2, IPS_1 and IPS_2 and identical logic values at RPS_3 and IPS_3. Three pairs of potentially equivalent compare-points are thus generated. In the next section, we will describe how to augment the search model circuit so that an ATPG program can be used to generate the partitioning vectors.



Partitioning Vectors
Partitions
I_1 I_2 RPS_1 RPS_2 RPS_3 IPS_1 IPS_2 IPS_3
[RNS_1 RNS_2 RNS_3 INS_1 INS_2 INS_3]
0 1 1 1 1 1 1 1
0 0 1 0 0 1
[RNS_1 RNS_2 INS_1 INS_2] [RNS_3 INS_3]
1 0 0 0 1 0 0 1
0 1 0 1 0 0
[RNS_1 INS_1] [RNS_2 INS_2] [RNS_3 INS_3]

Fig. 3 A partitioning example.

2.3 Use of ATPG for generating vectors

During the partitioning process, we need to generate partitioning vectors that sets opposite values between at least one reference compare-point and one implementation compare-point. Random pattern generation (RPG) can be used initially and followed by ATPG. In order to use an ATPG program to generate partitioning vectors, we need to augment the search model circuit to facilitate the generation process. Figure 4 shows the augmented search model circuit.

Gates *A* and D in figure 4 are n-input NAND gates. Gates *B* and *C* are n-input OR gates. Gates *E* and *F* are 2-input AND gates and Gate *G* is a 2-input OR gate. A test vector that sets a value 1 at Gate *G* will guarantee to set opposite values between at least one reference compare-point (RNS_i) and one implementation compare-point (INS_i). We can use an ATPG program on the augmented search model circuit to generate such vectors with the constraints $RPS_1 = \ldots = RPS_i = \ldots = RPS_n = IPS_1 = \ldots = IPS_i = \ldots = IPS_n$.

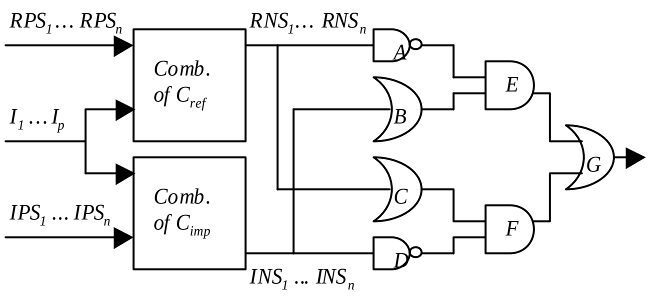


Fig. 4 Augmented Search Model Circuit for ATPG.

3. HANDLING OF DON'T CARES

Considering don't care conditions when comparing two designs is essential to avoid false negative verification results. Don't care conditions can come from the specification when performing RTL-To-Gate verification. They can also arise in hierarchical verification when external don't cares are derived from upstream modules for the verification of the downstream modules.

For our partitioning-based CPM method, there are two requirements for efficient handling of don't care conditions. First, we need to represent and model don't care conditions explicitly as part of the search model circuit so we can still readily apply an ATPG program to search for partitioning vectors. Second, the don't care representation circuit has to be size-efficient with respect to the original circuit. Our method of modeling don't care conditions is largely based on the work in [9].

3.1 Modeling don't cares

One way to represent an incompletely specified function that contains a don't care set is to use an interval. Given a function F with an on-set f and a don't care set d, we can represent the function with the interval $[f \,\&\, d', f + d]$. We interpret an interval $[m, M]$ as the set of functions $\mathcal{F} = \{f\colon m \subseteq f \subseteq M\}$. Any function in an interval is a valid implementation of the incompletely specified function f. A single don't care value x is represented by the interval [0,1]. Based on the interval concept, we can come up with a set of modification rules to change a circuit with don't cares to encode and represent explicitly the don't cares within the enhanced circuit.

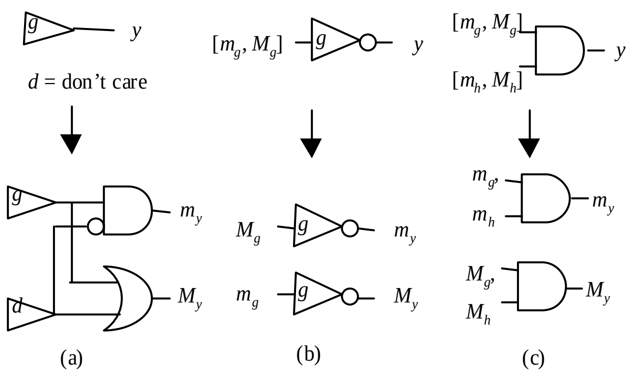


Fig. 5 Don't Care Circuit Modification Rules.

Interval-based circuit modification rules to model don't cares within a circuit structure are shown in Figure 5. For each gate or input port with an external don't care function d, we create two gates which implement the two endpoints of the equivalent interval as shown in Figure 5a. If the inputs of a gate have an interval associated with them, two new gates are created to represent the endpoints of the interval for the gate. Examples for an inverter and an AND gate are shown in Figure 5b and 5c respectively. Given a circuit with don't cares associated with some of its gates/ports, the maximum number of gates added to encode the don't care conditions in the original circuit is $O(n)$, where n is number of gates in the original circuit.

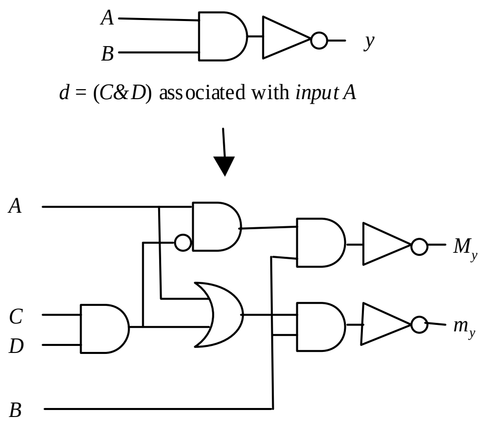


Fig. 6 An example for modeling don't cares.

An example to illustrate how to modify a circuit that has external don't cares associated with its primary input is shown in Figure 6. At the top of Figure 6 is the original circuit with 4 inputs A, B, C and D. There is a don't care condition $(C\&D)$ associated with the input A. The corresponding modified circuit encoded with the don't care information is shown at the bottom of the Figure 6.

The output y is split into two signals, m_y and M_y, that represent the two endpoints of the interval associated with it. If an input vector in the care set, e.g. C != 1 or D != 1, is applied to the primary inputs, the logical values at m_y and M_y are either all 0 or all 1. If any don't care vector, e.g. C = 1 and D = 1, is applied at the primary inputs, m_y and M_y are set to 0 and 1 respectively. By examining the logic values at the two corresponding interval endpoints for output y, we can determine whether an input vector is in the care/don't care set with respect to that particular output.

3.2 Impact on partitioning vector generation

Without loss of generality, we assume the reference circuit contains don't care conditions. In order to use an ATPG program to automatically generate partitioning vectors for circuits with don't cares, we first need to modify the original search model circuit (shown in Figure 1) based on the modification rules described in section 3.1. Two signals, RNS_{i1} and RNS_{i2}, are created for each compare-point, RNS_i, whose transitive fanin cone contains gates/ports with external don't cares. A vector that sets 0 and 1 at RNS_{i1} and RNS_{i2} respectively is a don't care vector with respect to the compare-point RNS_i. A care vector with respect to RNS_i will set either all 0 or all 1 at RNS_{i1} and RNS_{i2}.

Next, we need to modify the augmentation of the search model network. For every pair of RNS_{i1} and RNS_{i2}, we create one 2-input AND gate RNS_i_A and one 2-input OR gate RNS_i_O. The fanins to the two new gates are RNS_{i1} and RNS_{i2}. We replace the fanins of the n-input NAND gate A in Figure 4 with RNS_i_Os and replace the fanins of the n-input OR gate C with RNS_i_As. The augmentation circuit for a design with two reference compare points whose transitive fanin cones contain external don't cares is shown in Figure 7.

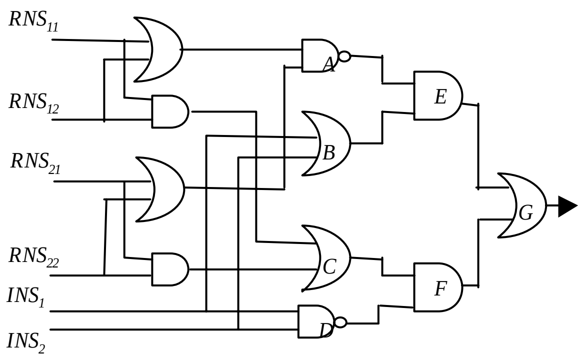


Fig. 7 Augmentation circuit for design with don't cares.

With the modifications described above, a generated vector that sets a value 1 at the output of the search model circuit will again guarantee to establish an inequivalent relationship between at least one reference compare-point and one implementation compare-point. However, multiple vectors may be needed to separate a partition into two as shown by the example in Figure 8. The first two vectors show INS_3 is not equivalent to RNS_1 and RNS_2. The last vector shows RNS_3 is not equivalent to INS_1 and INS_2. Based on the combined inequivalent information from the three vectors, we can then separate the partition into two.

Partitioning Vectors								Partitions					
I_1	I_2	RPS_1	RPS_2	RPS_3	IPS_1	IPS_2	IPS_3	[RNS_1	RNS_2	RNS_3	INS_1	INS_2	INS_3]
0	1	1	1	1	1	1	1	0	x	x	0	0	1
1	0	0	0	0	0	0	0	x	1	x	1	1	0
1	1	0	0	0	0	0	0	x	x	1	0	0	1

$\downarrow$

[RNS_1 RNS_2 INS_1 INS_2] [RNS_3 INS_3]

Fig. 8 A partitioning example with don't cares.

4. EXPERIMENTAL RESULTS

We performed experiments on 7 industrial circuits with sizes ranging from tens of thousands to hundreds of thousands gates. Table 1 shows the results obtained on a 450 MHz Sun UltraSPARC-III machine. Compare-point pairs in all test cases were successfully proved equivalent by the verification process that follows. The second column indicates the total number of gates in the reference designs. The third column in the table indicates the total number of potential compare-point pairs. The fourth column indicates the number of compare points matched by non-function-based methods. The fifth the number of compare points matched by our heuristic partition-based method. The last column indicates the number of CPU seconds it took to match by our heuristic method. There are two main factors contributing to the efficiency of our heuristic method. First, equivalence checking is completely avoided during the compare-point matching process. Second, existing matching information generated by non-function-based methods, that are usually run a priori as part of the matching process, are effectively used in our heuristic method.

Table 1 Experimental Results

DESIGNS	Num. of gates	Num.of compare-point pairs	Matched by non-function-based methods	Matched by our heuristic method	CPU time (sec)
1	140k	3583	2762	821	50
2	112k	1843	667	1176	45
3	71k	859	411	448	32
4	86k	646	369	277	29
5	40k	718	417	301	6
6	156k	1975	1815	160	28
7	364k	4378	1795	2583	176

5. CONCLUSION

We have described a heuristic compare-point-matching method based on a partitioning algorithm. We have shown how to use an ATPG program and a search model to implement the partitioning algorithm. Extension of the CPM method to handle designs with don't cares was also presented. Experimental results on large industrial circuits show the method is very efficient in practice.

6. ACKNOWLEDGMENTS

The authors would like to thank Jim Kukula, Stan Liao, Brian Lockyear, Lisa McIlwain, and Carl Pixley for their participation in many insightful discussions.

7. REFERENCES

[1] D. Brand, "Verification of large synthesized designs," in ICCAD, 1993, pp. 534 – 537.

[2] S.M. Reddy, W. Kunz and D.K. Pradhan, "Novel verification framework combining structural and OBDD methods in a synthesis environment," in DAC 1995, pp. 414 – 419.

[3] A. Kuehlmann and F. Krohm, "Equivalence Checking Using Cuts and Heaps," in DAC 1997, pp. 263 – 268.

[4] T. Filkorn, "Symbolische methoden für die Verifikation endlicher Zustandssysteme," Ph.D. dissertation, Institut für Informatik der Technischen Universität München, Munich, Germany, 1992.

[5] C.A.J. van Eijk and J.A.G. Jess, "Detection of Equivalent State Variables in Finite State Machine Verification," in IWLS, 1995.

[6] S-Y. Huang, K-T. Cheng, K-C. Chen and U. Glaeser, "An ATPG-Based Framework for Verifying Sequential Equivalence," in ITC, 1996, pp. 865 – 874.

[7] J.R. Burch and V. Singhal, "Robust Latch Mapping for Combinational Equivalence Checking," in ICCAD, 1998, pp. 563 – 569.

[8] H. Cho and C. Pixley. Apparatus and method for deriving correspondence between storage elements of a first circuit model and storage elements of a second circuit model. U. S. Patent 5,638,381, June 1997.

[9] T. Stanion. Circuit synthesis verification method and apparatus. U. S. Patent 6,056,784, May 2000.